Towards Correct-by-Construction Interrupt Routing on Real Hardware

Lukas Humbel, Reto Achermann, David Cock, Timothy Roscoe
Systems Group, Dept. of Computer Science, ETH Zurich

Abstract

In this paper we address the problem of correctly configuring interrupts. The interrupt subsystem of a computer is increasingly complex: a zoo of different controllers with varying constraints and capabilities form a network with limited connectivity. An OS which aspires to provable correctness must manage a limited set of interrupt vectors, delegate interrupts to device drivers and configure the controllers correctly. No well-specified approach exists.

As a foundation for applying language-level techniques like program sketching and synthesis to this problem, we present a formal model for interrupt routing which can capture all the system topologies and interrupt controllers we have encountered in the wild, show applications of such a model not possible with informal, ad-hoc approaches like DeviceTrees, and finally discuss an implementation based on the model which forms the new interrupt subsystem of the Barrelfish OS.

CCS Concepts • **Software and its engineering → Operating systems**; • **Theory of computation →** *Constraint and logic programming*;

Keywords Hardware configuration, Hardware abstraction, Interrupt routing, Eclipse/CLP

% HINT: do 'make submission' to remove the comments

1 Introduction

% Problem motivation, what are interrupts used for, contributions

We report on work to solve the problem of correctly (and provably so) configuring interrupt routing across a range of increasing diverse and complex hardware platforms. We present a formal model which can represent the complete interrupt topology (sources, vectors, links, controllers and cores) of real computer systems from a PC to a phone System-on-Chip (SoC) and capture the constraints on interrupt routing imposed by real-world hardware components.

We also show how to obtain properties of a given instance of the model, such as "Can every interrupt source be uniquely distinguished at its destination?", and to configure interrupt hardware to correctly route and deliver interrupts in the system based on operating system (OS) requirements. We also describe a concrete implementation based on the formalism which configures interrupt controllers on demand in the Barrelfish OS by realizing the model in Prolog.

The problem of correctly configuring the interrupt subsystem is surprisingly complex. As we show in Section 2, a modern computer includes many cores as potential destinations of interrupts and a complex network of interrupt controllers routing interrupt signals – it is not unusual for a signal to traverse more than 5 translation units before delivery to software. There also exists a wide variety of such controllers (we describe a representative set of 15 different ones), and support for virtualization adds further levels of complexity.

The problem is also important: a modern general-purpose OS has to handle the full complexity of a system like this, and the topology is generally not known when the OS is written. Correct operation of the system on a new piece of hardware depends on correct configuration of this network by software, and correct *reconfiguration* as device driver threads migrate and hardware is hot-plugged.

Moreover, the problem is not going away (and is therefore not amenable to a one-time hard-wired solution): new interrupt controllers are appearing all the time, and new SoC designs present new combinations of devices and heterogeneous cores with new constraints on which interrupts can be delivered where. Furthermore, formally *proving* the correctness on a system with interrupts must rest on a formal model of the underlying hardware.

Formally modeling interrupts also has value beyond system software design, since it can shed light on desirable properties of hardware designs (both complete platforms and individual controllers) as well.

This paper builds on our previous work, a formal model in Isabelle/HOL, on modeling memory and interrupt systems [1]. Our contributions over that paper are as follows: In section 3 we extend the model to capture constraints of real interrupt controllers, discuss how we have represented all the controllers we have seen to date in our systems, and

PLOS 2017, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144557

state useful properties of any given system that can be determined from the model. In section 4 we describe a practical application: We have implemented the model in Prolog and C and we instantiate it online to configure interrupt routing in the Barrelfish OS. We show how it can be used online to derive valid configurations for interrupt hardware.

In the next section, we further motivate the problem and delve into the complexity of modern interrupt systems.

2 Background

% There are many different controllers with various capabilities and constraints (table)

Modern interrupt hardware is complex. Whereas in the distant past, an interrupt was a dedicated electrical signal to the processor, today a computer has a network of interrupt controllers which can be configured to deliver many distinct interrupts generated by a given device to different vectors on different cores.

Table 1 shows 15 different interrupt controllers used by machines in our server room. New interrupt controllers are introduced all the time, whether evolutions of existing designs or new, specialized functions for particular SoCs. Each has different capabilities and constraints on the number of interrupt signals they can source and sink, and how they can map between them. For instance, the venerable Intel 8259A PIC [17] has a fixed mapping of 8 input ports to a single output port and 8 bit vector. The Local APIC [19] maps interrupt messages on a bus to a corresponding local core vector. Intel IOAPIC controllers [18, 20] convert events directly from PCI functions or through PCI Link Devices [28] to APIC messages in a particular delivery mode, but behave differently when combined with an IOMMU [22], which can translate memory writes from devices to message-signaled interrupts [28]. The ARM GICv2 [3] supports 1024 different interrupts but not all can be delivered to all cores, and vectors cannot be changed. The compatible CoreLink GIC-400 [2] adds additional constraints on its reconfigurability, the GICv3 exists in two variants [4] with implementation-defined limits on vector size and GICv4 adds virtualization support [4]. Additionally, the ARM GIC are programmed using a memory mapped register and/or CPU interface.

% Hard to configure those things

These controllers are connected in a non-trivial platform-specific network. Figure 1 shows a simplified PC-based illustration. Interrupts may be delivered to a single core, a set (1-N) or broadcast. Virtualization allows interrupt delivery directly to a virtual machine [21, 22].

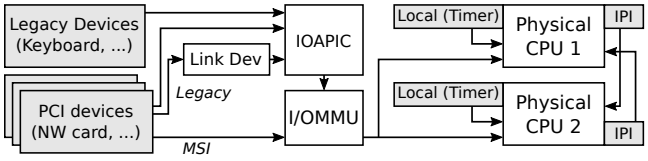

Figure 1. Simplified x86 interrupt network

The OS must discover and correctly configure this network dynamically. Some topology data can be obtained from PCI discovery [28] and ACPI [34], but it is incomplete or may not exist at all. DeviceTree [10] files are used by many OSes to work around this, but the file format has no clear semantics, is error-prone [30], and despite containing controller information [29] fails to capture configuration constraints or cover inter-processor interrupts. Even so, the proliferation of DeviceTrees shows that configuration is a problem.

After discovery, correct configuration of a modern computer is essentially a network routing problem with highly constrained switches, but current OS designs reflect a legacy of much simpler hardware.

% Other OSes deal with this by hardwiring and non-uniform interfaces (depending on the configuration)

Linux, for example, defines a single namespace of "IRQ numbers" for all interrupts, and then attempts to map this to a strict hierarchy of interrupt controllers. "IRQ Domains" [25] map Linux IRQ numbers to hardware sources and implicitly hard-codes the topology. Device drivers are responsible for identifying the controllers they need to program (via a driver interface) to deliver interrupts correctly. The common case is to deliver an interrupt to all cores, and vector numbers are assumed to be the same across all cores. Constraints in interrupt routing are not well handled and generally special-cased in the code.

Chen *etal.* [9] verify an interruptible operating system kernel including a simple verified interrupt controller driver. The focus of our work is on the topology of the interrupt system, we are interested in properties of the configuration and ensure, for instance, that the correct controller is configured.

Stepping back, a better approach is to define a formal model which captures the complexity of modern interrupt subsystems and provides both a basis for verifying implementations and a template for engineering a correct solution which works across a wide variety of platforms. This paper describes early work in this direction: both a preliminary model and an implementation.

3 Model

% start with that this is an extension to the MARS paper

We base our model on prior work [1] about formally specifying memory accesses and interrupts and extend it to enable interrupt controller configuration. Currently, the model is implemented informally in Prolog, described in section 4. We present the extensions necessary to provide a formal basis for that implementation.

% briefly describe the previous work

We express the topology of a system as a *decoding net*, a directed graph consisting of nodes with two properties: *i)* a set of *accepted* addresses *ii)* a set of *translated* addresses

Controller	In Port #	In Vector Size	Out Port #	Out Vector Size	Constraints
PIC	$n \cdot 8 + (8 - n)$	0 bit	1	8bit	fixed
I/OAPIC	24	0 bit	16	8 bit	None
I/OxAPIC	24	0 bit	256	8 bit	None
I/Ox2APIC	24	0 bit	2^{32}	8 bit	None
I/Ox2APIC + I/OMMU	24	0 bit	2^{16}	0 bit	None
LAPIC LVT	7	0 bit	1	8 bit	None
PCI Link Device	4	0 bit	4	0 bit	None
MSI Link Device	2^{0-4}	0 bit	2^{32}	32 bit	Same port, contiguous addresses
MSIx Link Device	64 – 2048	0 bit	2^{32}	32 bit	None
IRTE Mapper	2^{20}	32 bit	1	16bit	fixed
IPI RT	?	0 bit	16	8 bit	None
I/OMMU	1	16 bit	2^{32}	8 bit	None
ARM GICv2 Dist	987	0 bit	8	10 bit INTID	out port == in port
ARM GICv3 ITS	2^{32}	32 bit	1	10 bit INTID + 16 bit ICID	unique INTID outputs
ARM GICv3 CT	1	10 bit INTID + 16 bit ICID	2^{32}	10 bit INTID	INTID must match input INTID

Table 1. Characteristics of interrupt controllers showing input and output port numbers and vector sizes.

Definition (Port Set). $\mathbb{P} \subset \mathbb{N}$

Definition (Interrupt Format).

$$\mathbb{I} = \{Empty, Vector, Mem\}$$

Where

- $Empty = \{\}$ is an interrupt with no associated data.
- $Vector \subset \mathbb{N}$ is the set of interrupts that can be described using a single interrupt vector number.
- $Mem \subset \mathbb{N} \times \mathbb{N}$ the set of memory write operations represented as address-data word tuples.

Definition (Mapping Function). Partially defined function from an input to an output format-port tuple.

$$\mathbb{F} :: \mathbb{I} \times \mathbb{P} \rightharpoonup \mathbb{I} \times \mathbb{P}$$

Definition (Controller).

$$C = (inPorts, outPorts, mapValid) \in \mathbb{P} \times \mathbb{P} \times 2^{\mathbb{F}} = \mathbb{C}$$

Where $2^{\mathbb{F}}$ denotes all possible mapping functions and $mapValid \subseteq 2^{\mathbb{F}}$ are the valid mapping functions. $\mathbb{C}$ is the set of controllers.

Definition (Configuration).

$$Conf :: \mathbb{C} \rightharpoonup \mathbb{F}$$

A configuration is valid if $\forall C.Conf(C) \in C.mapValid$.

Definition (System).

$$\mathcal{S} = (inPorts, outPorts, ctrls) \in (\mathbb{P} \times \mathbb{P} \times \mathbb{C})$$

Where $inPorts$ and $outPorts$ are the sets of incoming and outgoing ports respectively $ctrls$ is a set of controllers.

Figure 2. Interrupt Model Definition

that map onto another node, where addresses here represent interrupt ports. Address resolution starts at a particular node and address and terminates if a node accepts the input address or it is not in the set of translated addresses.

3.1 Model refinement

% show the delta to the decoding net model

The nodes are a set of interrupt sources (e.g. devices), a set of destinations (e.g. an interrupt vector on a core) and a set of interrupt controllers. We refer to addresses on nodes in the decoding net as *ports*. We assign a globally unique identifier to all ports. We further extend the model in [1] with the refinements below and summarize the extensions in Figure 5.

Ports and Vectors: In a plain decoding net, a node only knows about addresses. Interrupt controllers, in contrast, can distinguish between source (i.e. *port*) and actual data (i.e. *vector*) transferred. Since ports and vectors are both of finite domain, we could define a mapping of (port, vector) pairs to a single numeric range through enumeration. However, since the separation into *ports* and *vectors* naturally reflects the exposed programming interface to interrupt controllers, we keep them separate in the refined model.

This does lead to redundancy in the model and a potential choice in how to split the representation of a given controller between ports and vectors. We use the following rule of thumb: Multiple output ports should be used if the controller can direct interrupts to multiple destinations. Similarly, multiple input ports should be used if the controller can distinguish between different interrupt sources. Vectors should be used to distinguish between input events from the same source, or outputs to the same destination.

Interrupt formats: The controllers in Table 1 use three different interrupt formats: *i)* a *plain signal* asserting an occurred event (e.g. device interrupt), *ii) plain signal + vector* providing a word of information about the event (e.g. CPUs receive an interrupt vector) and *iii)* a *memory write* of a data word to a specific address (as in MSI-X). Therefore, the node's translate function must be able to differentiate and convert between different interrupt formats.

Configurability: One of our goals is to *correctly* configure interrupt controllers, thus we need to capture the set of possible configurations of a controller. We add a third property, the set of *valid translate functions*, to the decoding net nodes that expresses supported interrupt formats, data words, and transformations.

We further define an *interrupt system* as a tuple of incoming ports, outgoing ports and controllers, i.e. a complete

decoding net. To express the current state of the system, each controller is assigned a *configuration*. We say that the configuration is *valid* if each controller is assigned a valid translation function.

Note the model allows two controllers that produce/consume different interrupt formats to be linked. For consistency, we interpret this as stating that the controllers cannot receive messages from each other.

RA: ideally: show that this is a refinement of the decodig net model.

% Expressing currently allocated interrupt routes

% > - how do we represent the hardware / configuration of the system

LH: There should be a principle for the case how to choose port numbers. For instance the outport zero of a PIC triggers the vector 32 at the CPU. There should be some principle how to deal with this situation. Or maybe its also a consequence of the preference of fixed/-fully dynamic principle

3.2 Representing interrupt controllers

We have expressed all the interrupt controllers in Table 1. Note there is no unique representation of an interrupt system: identifiers of ports and vectors can be changed while preserving the controller's semantics and even splitting and merging of controllers is possible – it is theoretically possible to express an entire interrupt system using a single controller and a complex predicate for valid mappings. Practical considerations of modularity, reuse, and readability determine a "good" representation; we give three examples (Figure 3).

On the **Intel IOAPIC** all of 24 input ports are directly connected to an interrupt source. The interrupt format is a plain signal. Each port can be configured independently, and interrupts are always delivered on the APIC bus with a vector in the range [32, 255]. We model it with 24 input ports because of the direct input connections and provide a valid mapping function to constrain the possible emitted vectors.

Devices supporting **Message Signaled Interrupts** (MSI) can trigger up to 32 different interrupts. We model such sources as interrupt controllers themselves since they determine the delivery destination of interrupts. As with the IOAPIC, we express the device as a controller with 32 plain-signal input ports that generate consecutive memory writes. These writes (the MSIs themselves) impose dependencies between different "ports" on the device, and so we need to carefully constrain the set of valid configurations for a MSI-capable device.

As a final example, the **Intel IOMMU** translates all MSI memory writes into an index into a single table, using a non-injective function. While we could represent this constraint logically in the model, it is simpler and more elegant to split the IOMMU into two imaginary controllers: a fixed function called IRTEMAP which maps the MSI into an integer on a single output port, and the remapping table called IRTE with multiple output ports that captures the routing functionality. The IOMMU cannot perform a different routing decision based on the source, therefore we use one input port.

We have found this trick of splitting a controller into a fixed-mapping controller and a freely configurable one which as a pair preserve the original semantics to be useful and quite widely applicable: it pushes complexity out of the constraints and into the model, and as a side effect simplifies implementing the controller drivers themselves.

However, not all controllers can be split. Mapping constraints that depend on the configuration of other ports, such as in the case of the MSI controller, can not be split.

3.3 Useful properties

% What can we do with this model? model properties, splitting of controllers (proof)

LH: I moved all paragraphs that concern "how we model real systems" into Expressing interrupt controllers along with the examples.

Once we have a model that can capture both the topology of a real machine's interrupt subsystem and the functionality of its programmable interrupt controllers, we can start to formulate useful properties of a given system that can be proved (or disproved) from the model representation.

A first case is **Reachability**. It is particularly the case with SoCs that a given interrupt cannot be delivered to any processor in the system. Using the model, we can derive the reachability matrix for interrupts in a given system. Furthermore, since our model also integrates configuration, we can also answer a slightly more challenging question: *Given a set of interrupt source-destination pairs, is it possible to find a configuration that connects all of them?*

For each source-destination pair, there exist multiple controller configurations that connect these two parts and devices can use different signaling mechanisms, which in turn may result in multiple distinct routes through the controller network. As long as controllers can distinguish incoming interrupts, the intermediate representation does not matter (e.g. IOMMU + MSI). Using the model, we can enumerate all possible configurations.

A second useful property of a system is **reliable delivery**: under what conditions can be guaranteed that every interrupt will arrive at the appropriate destination. This may be required to prove the liveness of the system, but even if not (where interrupts are a "hint" to improve the performance of a polling model), failure to deliver interrupts can create hard-to-diagnose performance degradation.

Note that this subsumes the problem of verifying whether a given configuration of the system is "correct" but is stricter: it includes the idea that at no point during a *reconfiguration* of the interrupt system will it enter a state where interrupts can be lost or misrouted. Given a system representation in

IOAPIC

$C_{IOAPIC} = (inPorts, toCPUs, apV)$ with $|inPorts| = 24, \quad f \in apV \leftrightarrow \forall port : f(\_, port).int \in \{32..255\}$

Intel IOMMU

$C_{IRTEMAP} = (inPorts, IRTE, irV)$ with $|inPorts| = 1, \quad irV = \{f(mem(addr, data), port) \rightarrow \{addr + data, \_\}\}$

$C_{IRTE} = (IRTE, toCPUs, 2^{\mathbb{F}})$ with $|IRTE| = 1$

MSI

$C_{MSI} = (inPorts, memWrite, msiV)$ with $inPorts = [0, 32], \ base_{addr} = \text{constant}, \ base_{data} = \text{constant}$
$f \in msiV \leftrightarrow f(in, port) \rightarrow \{mem(base_{addr}, base_{data} + port), \_\}$

Figure 3. Real interrupt controllers expressed in model

our model we can verify the correct delivery of each interrupt for any given configuration of interrupt routers.

A less serious but dual problem is spurious interrupts. Our model can be used to constrain the set of possible causes of a spurious interrupt received at a given core, as long as our model of each controller is sufficiently faithful.

As a final example, **distinguishing interrupts** is an important requirement for an OS so that it can invoke the appropriate device driver. This should be trivial (each distinct interrupt should arrive on a different vector on a given core), actually *proving* that it is the case is not, and has some similarities with a network capacity problem. A controller is able to distinguish up to $N = \#ports \times \#vectors$ different interrupts where each ports-vector tuple identifies an entry in the controller's routing table. If there are more interrupt sources than the smallest interrupt controller can distinguish, interrupt *sharing* (two distinct sources trigger the same destination) may occur eventually – when using a suboptimal configuration heuristic even before all port-vector tuples have been allocated. The model can identify which interrupts are shared and where. **Thanks to the inclusion of configuration options we can pick a minimal sharing configuration. Currently used heuristics fail to do so on complex machines.**

4 Implementation

The paper mentions the possibility of connecting the formal work for interrupt routing with verified operating system kernels, but it does not contain any discussion how such connection can be established. Without this connection, it is difficult to see how those properties described in Section 3.3 will actually be used.

Also, timer interrupt and IPI are somewhat special since they are used to support real-time behaviors and concurrency. Are they treated any differently in your model? Will they require new properties other than those listed in Section 3.3?

We used our model to entirely replace the existing interrupt subsystem of the Barrelfish OS [6]. In Barrelfish, our work is made easier by the System Knowledge Base (SKB) [31] – a Prolog engine and constraint solver – which holds the state of the model as a set of Prolog facts and predicates, and implements the routing algorithm.

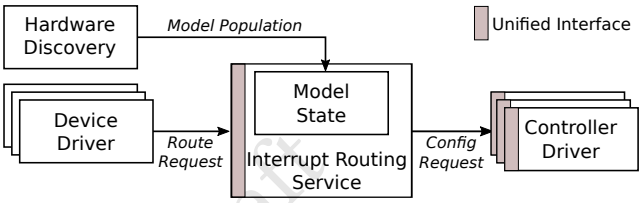

Figure 4. System Architecture

At time of writing, the implementation successfully configures device interrupts on demand on all real and virtual hardware used by the Barrelfish development team, including a variety of x86 and ARMv8-based server machines and ARMv7-A development boards.

While the current implementation is based on the formal model represented in logic programming, it does not provide the assurance of a fully-verified implementation, and the low-level hardware access for discovery and register programming is hand-coded in C and Barrelfish's (non-verified) Mackerel domain-specific language for hardware [33].

Nevertheless, the implementation is functional, demonstrates the viability of the approach, and has greatly simplified and unified device programming across diverse platforms in Barrelfish. Moreover, it is clear that a different (perhaps more complex) implementation is possible in C for monolithic kernel systems like Linux.

Our implementation consists of 179 lines of Prolog for the generic model. As an example, 185 additional lines of Prolog implement the x86 specific part; the bulk of the latter dealing with populating the model with topology information discovered from ACPI and the (user-space) PCIe driver. For a given interrupt controller, the constraints on routing it imposes can usually be expressed in a single line.

The high-level architecture is shown in Figure 6.

4.1 The routing service

% > - algorithms on top of the model: what do we do

% > - explain how a route from source ot destination can be found, accounting for current allocations and controller constraints

The *Interrupt Routing Service* (IRS) is implemented inside the SKB as a set of inference rules (analogous in this case to stored procedures in a relational database) and executes the

routing algorithm incrementally over the SKB's representation of the interrupt topology and current configuration. The output of the algorithm is a set of (re)configurations for specific interrupt controllers, which are then programmed by their respective driver processes.

Barrelfish's Multikernel [8] architecture, as in a microkernel, implements most drivers in user space, including most of the interrupt controller drivers. The IRS model encodes the routing constraints specific to particular interrupt controller types, but the interface to an interrupt controller used by the IRS can be entirely generic, simplifying implementation.

The model contains all information necessary to route interrupts. A routing request consists of an interrupt source and an interrupt destination. For simplicity, we assume the interrupt destination is provided by the requester. Often, the interrupt destination has some freedom: It is important on which CPU the interrupt ends up, but the exact vector triggered is not important.

The routing algorithm determines a valid configuration for each controller such that all the existing routes plus the new request are satisfied. A natural, though inefficient, approach in Prolog is a back-tracking, depth-first search. The entering interrupt to be routed is followed to its first controller, the first configuration that does not discard the interrupt is considered, tracing the interrupt to the next controller and repeating it until a destination is found. If the interrupt destination does not match, or it discards one of the existing routes, we backtrack. In practice, we can improve this by only picking output ports that get us closer to the desired destination, and failing (or resorting to interrupt sharing) as soon as we encounter a link where the new interrupt doesn't "fit" in the identifier space. This latter event is rare, and only occurs in highly resource-constrained systems.

So far, solving time has had a negligible impact on system performance, even on complex multisocket platforms.

4.2 Topology discovery

% > - what does it need to do a discovery? (can everything be discovered)

Various sources of hardware discovery populate the model for a given machine. Existing drivers for ACPI and PCIe in Barrelfish were simple to modify for this purpose, since they already entered discovered information in the SKB, indeed, in some cases a single Prolog rule provided an appropriate "view" over existing information.

Ideally, PCIe, ACPI, etc. would allow the OS to discover the entire interrupt topology online. However, many platforms (in particular, ARMv7-A SoCs), completely lack a discovery mechanism for devices and interrupts. In other cases, discovery is incomplete for one or more reasons (such as devices which are not on a discoverable bus). For non-discoverable interrupt information we fall back to static information in compiled Prolog files provided in the startup RAM disk. The OS loads at boot the relevant predicates based on what system (and core) it is booting on.

+ S4.2, "...we fall back to static information in compiled Prolog files provided in the startup RAM disk." Can this information be checked at run time? When topology cannot be discovered, perhaps it can at least be tested.

Finally, even the information gained from a discovery mechanism is usually insufficient to instantiate the model, even if all the controllers are discovered. The topology itself is often represented only implicitly, such as through the hierarchy of the PCI bus. Often, certain links or translations are missing (for example, in ACPI, it is not discoverable how MSI interrupts are translated to CPU vectors). For this information, we also fall back knowledge the system programmer has extracted from datasheets (or, conceivably, DeviceTree files) and coded into the configuration algorithm or a supplemental Prolog file. **Similarly, the paper could benefit from more discussion of whether the knowledge needed to use this is readily available. It seems like some information needed is not readily documented (section 4.2) - hence raising the question of whether that information is needed in practice, or just needed to make the model work? This information is crucial for any operating system. Our approach explicitly exposes all translation units, while in commodity systems, this knowledge is implicitly contained in program code.** Note that the topology and set of controllers in the system can be entirely dynamic.

4.3 Clients

Clients of the IRS are device drivers which wish to receive interrupts from the devices they manage. A full description of the Barrelfish driver protocol (including the authorization framework for interrupts) is beyond the scope of this paper, but can be briefly summarized as follows.

When a device driver is started by the Barrelfish device manager, it receives *capabilities* for resources it needs to access the device. This includes memory-mapped I/O register areas but also capabilities granting the right to receive interrupt notifications from a specific interrupt source. The driver creates a communication endpoint (also represented by a capability) and hands this together with the interrupt source capability to the IRS. Capabilities are also used to grant access to specific vectors in interrupt controllers (up to and including interrupt delivery vectors on destination cores). This allows more decentralized implementation in the future, but crucially isolates the interrupt resources of a device and its driver from others in the system.

4.4 Discussion

% > - the unified interface

Our implementation was driven both by the formal model and the particular architecture and facilities of Barrelfish.

However, the separation of mechanism from policy (routing) that results in, we claim, an elegant solution and we see no strong reason why the techniques are not equally applicable to monolithic systems like Linux or microkernels like seL4.

The approach allows high-level language techniques (like inference and constraint solving) to be applied to low-level concerns (interrupts), with a consequent simplification both of individual drivers for peripheral device and interrupt controllers, and also the core of the OS as a whole. A further benefit is that generic interfaces to interrupt controllers and IRS do not end up in contradiction with the behavioral quirks of specific components.

5 Ongoing work and conclusion

Two major aspects of interrupts are not yet fully captured by our model. The first is that interrupts are currently unicast: we configure interrupt controllers to forward an interrupt to *one* destination, rather than multicasting (or broadcasting) the interrupt to many destinations. This is well-suited to the Barrelfish architecture, but less so for a monolithic system like Linux, and in any system is valuable for, e.g., optimizing TLB shootdowns within a shared physical address space. Extending the model to support multicast is straightforward.

Secondly, we do not address dynamic aspects of interrupt delivery, such as how interrupts are acknowledged, and the distinction between edge- and level-triggered interrupts. While rare these days, level-triggered interrupts have important use-cases, and how to capture the distinction is a topic of ongoing work.

Nevertheless, we have devised a model of interrupt delivery and formulated it, together which descriptions real hardware platforms and components in Prolog and demonstrated its practicality in a real OS. In previous work we have shown how a similar model can be formalized in Isabelle/HOL [27], and we plan on fully formalizing the propsed model.

To avoid manual translation between the dual Prolog and formal representation, we plan to extend a concrete syntax and compiler written to express complex memory subsystems [32] to include interrupt topologies and controller configurations. Then have this language generate both Prolog facts for runtime use and a formal representation for offline reasoning.

A practical extension would be to compile a DeviceTree file into this syntax widen our device support. However, we have found that the lack of clear DeviceTree semantics still requires a manual (human) step in the translation process to a formal specification.

Our programming code works on static snapshots of the interrupt subsystem, but does not address how to get from one configuration to a new one. We are exploring *program synthesis* techniques to generate a series of atomic reconfiguration operations that can, by construction, reconfigure the interrupt subsystem so that at no point does it pass through a "bad" state (e.g. where interrupts are lost or misdelivered).

+ S5, "...generate a series of atomic reconfiguration operations that can, by construction, reconfigure the interrupt subsystem so that at no point does it pass through a 'bad' state...." If this discussion remains in the final paper, it might be worth citing similar work that does this sort of thing in the context of SDNs: e.g., Reitblatt et al., "Abstractions for Network Update," SIGCOMM '12, <https://doi.org/10.1145/2342356.2342427>; and Ghorbani et al., "Transparent, Live Migration of a Software-Defined Network," SoCC '14, <https://doi.org/10.1145/267 I offer these only as examples, not as a canonical works.

We are also exploring program synthesis for programming individual interrupt controllers. In particular, by expressing the hardware registers and their meanings in the form of a *program sketch*, we can use synthesis techniques to generate correct register operations on each device.

Our work is at an early stage, but our experience both with the formal and implementation aspects suggests that we have a solid foundation for our ongoing work, with the long-term goal of generating correct and efficient OS code for an increasingly complex hardware landscape.

References

[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2017. Formalizing Memory Accesses and Interrupts. In *2nd Workshop on Models for Formal Analysis of Real Systems (MARS 2017)*. Electronic Proceedings in Theoretical Computer Science, Uppsala, Sweden, 66–117. https://doi.org/10.4204/EPTCS.244.4

[2] ARM Ltd. 2011. *CoreLink GIC-400 Generic Interrupt Controller - Technical Reference Manual* (revision r0p0 ed.). ARM.

[3] ARM Ltd. 2016. *ARM Generic Interrupt Controller - Architecture version 2.0* (issue b ed.). ARM.

[4] ARM Ltd. 2016. *ARM Generic Interrupt Controller Architecture Specification - GIC architecture version 3.0 and version 4.0* (issue c ed.). ARM.

[5] John H. Baldwin. 2007. PCI Interrupts for x86 Machines under FreeBSD. In *Proceedings of the 4th Technical BSD Conference (BSDCan '07)*. FreeBSD Foundation, Ottawa, USA, 17. https://people.freebsd.org/~jhb/papers/bsdcan/2007/article.pdf

[6] Barrelfish team. 2017. The Barrelfish Research Operating System. (August 2017). www.barrelfish.org.

[7] Richard Barry. 2016. *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide* (pre-release 161204 ed.). Real Time Engineers Ltd.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, Big Sky, Montana, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[9] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, Santa Barbara, CA, USA, 431–447. https://doi.org/10.1145/2908080.2908101

[10] Devicetree.org. 2016. *Devicetree Specification* (release 0.1 ed.). Linaro, Ltd. http://www.devicetree.org/specifications-pdf.

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

[12] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. 2008. Certifying Low-level Programs with Hardware Interrupts and Preemptive Threads. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, Tucson, AZ, USA, 170–182. https://doi.org/10.1145/1375581.1375603

[13] Thomas Gleixner and Ingo Molnar. 2010. *Linux generic IRQ handling*. The Linux Foundation. https://www.kernel.org/doc/htmldocs/genericirq/.

[14] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

[15] Gang Hou, Kuanjiu Zhou, Junwang Chang, Rui Li, and Mingchu Li. 2013. *Interrupt Modeling and Verification for Embedded Systems Based on Time Petri Nets*. Springer Berlin Heidelberg, Berlin, Heidelberg, 62–76. https://doi.org/10.1007/978-3-642-45293-2_5

[16] Yanhong Huang, Yongxin Zhao, Shengchao Qin, and Jifeng He. 2015. Probabilistic Denotational Semantics for an Interrupt Modelling Language. In *Proceedings of the 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS) (ICECCS '15)*. IEEE Computer Society, Washington, DC, USA, 160–169. https://doi.org/10.1109/ICECCS.2015.35

[17] Intel Corporation. 1988. *8259A - Programmable Interrupt Controller*. Intel Corporation. Order Number: 231468-003.

[18] Intel Corporation. 1996. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*. Intel Corporation. Order Number: 290566-001.

[19] Intel Corporation. 1997. *MultiProcessor Specification* (revision 006 ed.). Intel Corporation.

[20] Intel Corporation. 2014. *Intel 64 Architecture x2APIC Specification*. Intel Corporation. Reference Number: 318148-004.

[21] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual* (volume 3, systems programming guide ed.). Intel Corporation.

[22] Intel Corporation. 2016. *Intel Virtualization Technology for Directed I/O - Architecture Specification* (revision 2.4 ed.). Intel Corporation.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, Big Sky, Montana, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[24] Xin Li, Yanhong Huang, Jianqi Shi, Jian Guo, Huibiao Zhu, and Yuanmin Xu. 2014. pIML – An Interrupt Program Modelling Language for Real-Time and Embedded Systems. In *Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 01 (APSEC '14)*.

IEEE Computer Society, Washington, DC, USA, 78–85. https://doi.org/10.1109/APSEC.2014.21

[25] Grant Likely, Linus Walleij, Jiang Liu, Jianyu Zhan, Marc Zyngier, Kevin Cernekee, Xishi Qiu, and Mark Brown. 2016. *irq_domain interrupt number mapping library*. The Linux Foundation. https://www.kernel.org/doc/Documentation/IRQ-domain.txt.

[26] Ingo Molnar and Max Krasnyansky. 2013. *SMP IRQ affinity*. The Linux Foundation. https://www.kernel.org/doc/Documentation/IRQ-affinity.txt.

[27] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. 2017. Isabelle / HOL Proof Assistant. (August 2017). http://isabelle.in.tum.de.

[28] PCI Special Interest Group. 2004. *PCI Local Bus Specification Revision 3.0* (revision 2.3 ed.). PCI Special Interest Group.

[29] Thierry Reding, Rob Herring, Grant Likely, and Bjorn Helgaas. 2014. *Specifying interrupt information for devices*. Kernel.org. https://www.kernel.org/doc/Documentation/devicetree/bindings/interrupt-controller/interrupts.txt.

[30] Mark Rutland. 2013. Device Tree - The Disaster So Far. Online. (2013). ELC Europe. http://elinux.org/images/8/8e/Rutland-presentation_3.pdf.

[31] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. 2011. A Declarative Language Approach to Device Configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, Newport Beach, California, USA, 119–132. https://doi.org/10.1145/1950365.1950382

[32] Daniel Schwyn. 2017. *Hardware Configuration with Dynamically-Queried Formal Models*. Master's thesis. Systems Group, ETH Zurich.

[33] Timothy Roscoe. 2013. *Barrelfish Technical Note 2 - Mackerel User Guide* (version 1.5 ed.). Barrelfish Project.

[34] UEFI Forum. 2017. *Advanced Configuration and Power Interface Specification* (version 6.2 ed.). UEFI Forum.

[35] Yongxin Zhao, Yanhong Huang, Jifeng He, and Si Liu. 2011. Formal Model of Interrupt Program from a Probabilistic Perspective. In *Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '11)*. IEEE Computer Society, Washington, DC, USA, 87–94. https://doi.org/10.1109/ICECCS.2011.16

A Appendix

B Introduction

RA:

1. **Introduction: we show how we can use prolog and constrain solvers to program the interrupt controllers in the Barrelfish OS**
2. **what's different? / related work**
3. **Example system and how we can represent it in our model**
4. **show the model implementation in Prolog and Barrelfish**
5. **evaluation that it works and is efficient**
6. **future work**

RA: this needs some references.. and it's a bit x86 heavy LH: I'm not sure if we want to do the history review here. Maybe we want to motivate the problem with a couple of given concrete examples challenges. Like: a single MSIx device can trigger more interrupts than a X86 core can distinguish. and ...

Interrupts are an asynchronous notification mechanism for events that occur in a system. The most prominent class being device interrupts. Device interrupts are triggered by devices upon a specific condition. For instance, a network interface card may trigger an interrupt on reception of a network packet. Devices, like a network card, may be accessible among multiple CPU cores and interrupt can be delivered to one of those CPUs. But also CPU specific devices exist. For instance, the operating system uses timer interrupts for scheduling. Typically these timer device is private to a CPU. It can only be programmed and configured from the owning core. Also, the interrupt from this device can only be received by that core.

LH: I think the following paragraph is (contentwise) OK... In the past, the interrupt subsystem used to be rather simple: First, there was only one core. Second, the configuration of the platform was, for a large part, fixed. Most of the interrupts had a particular event associated to it. For instance interrupt vector 1 was the keyboard. A small number were left unassigned to support pluggable devices. Each interrupt had a dedicated physical line. Since CPUs only have few physical interrupt lines, a interrupt controller was introduced. The job of the interrupt controller is to multiplex multiple physical interrupt lines to the few wires going into the processor. On interupt arrival, the CPU queried the interrupt controller to determine which interrupt has occured. The interrupt controller in this scenario is purely a multiplexer and did not offer any configuration options, such as changing the event to interrupt number mapping, to the operating system.

The limited number of those lines resulted in interrupt sharing. In absence of any discovery mechanism and with limited configuration options the device to interrupt mapping was static.

LH: Next paragraph i dont like. too long/too detailed...
With the introduction of multiprocessor systems, there is no longer a single destination for interrupt delivery. The advanced programmable interrupt controller (APIC) system provides a programmable bus for interrupt delivery supporting unicast, multicast, and broadcast interrupt delivery. Similar systems have been introduced on other platforms, like the generic interrupt controller (GIC) system for ARM.

Each processor is a client on this bus and is able to trigger interrupts to other processors using interprocessor interrupts (IPI). Devices now trigger interrupts through an IO interrupt controller which is another client on the interrupt bus translating device events to bus messages. Discovery mechanisms were added to query the topology and connectivity of interrupt controllers.

LH: Next paragraph i dont like. too long/too detailed...

The PCI Express bus organizes devices into a hierarchy. The devices no longer have a dedicated interrupt line to the interrupt bus. PCI bridges and PCI link devices can shuffle the interrupt lines either in a configurable or fixed way. Message signaled interrupts (MSI/MSI-X) overcome the limit of physical interrupt lines by using memory writes to signal interrupt conditions.

To support high performance hardware virtualization, it is necessary to map physical devices directly into virtual machines. Hardware devices using DMA are programmed using physical memory addresses but a virtualized operating system is unaware of physical addresses, it can not by itself program such a device correctly. Hence, vendors added hardware support to make this mapping possible. Hardware supported virtualization allows direct and safe pass-through of devices into virtual machines using IOMMU. As the acronym indicates, the main task of this hardware unit is to provide memory address translation for memory transaction origination from IO devices. Which means, IO devices do not have to programmed using physical addresses anymore. The same problem as with physical addresses exists with interrupt vectors, since the guest assumes he has exclusive control of the system, he might create conflicting mappings with other guests or with the hypervisor. To resolve this, IOMMUs allow also to remap interrupts. Most hypervisors receive the interrupts and inject a virtual interrupt in the guest, although more efficient techniques have been described **LH: insert citation**.

The IOMMU system described allows a all-or-nothing mapping of physical devices to virtual machines. If sharing is desired, the hardware device has to support single root I/O virtualization (SR-IOV). This creates new, logical devices from a single physical device each of which can trigger interrupts. To improve interrupt delivery efficiency into virtual machines, hardware support has been added to alleviate the need to involve the hypervisor (interrupt posting). At this point, the processors local interrupt controller has been virtualized, where the bus itself is still emulated by software.

The modern interrupt features such as MSI-X and virtualization extensions greatly increase the configuration space and number of possible interrupt sources in a system. For instance, using MSI-X a single device can trigger up to 2048 different interrupts. On the other side, a single X86 core can distinguish at most 256 different interrupt sources. Despite this more powerful delivery mechanism, legacy interrupts are unlikely to dissappear in the near future. To make them dissappear, every interrupt triggering device must be able to issue memory transactions, which seems unlikely for small devices such as core local timers.

The operating system therefore needs to understand the interrupt controller topology, which is apart from fixed knowledge about the platform, done throught discovery. It must further allocate resources such as interrupt vectors and configure the controllers and destination vectors accordingly. All of this with taking different interrupt models and different interrupt controller constraints into account. We present a selection of controllers with their constraints in section C.3 and give an overview of more controllers in Table 1).

In this work, we propose to decouple the discovery, description, configuration and controller drivers in order to tackle the complexity of the interrupt subsystem. This separation of allocation and routing decisions from the actual topology and controller drivers enables to specify policies and objective functions independently. We implemented this architecture in the Barrelfish research OS and demonstrate that our approach is applicable in a diverse set of different architectures and platforms. Our contributions are:

- An formal model of the interrupt subsystem close to hardware.
- A novel architecture to represent and program the interrupt subsystem.
- Generalized routing and configuration algorithms.
- An implementation of our model in C and Prolog.

C Background and Related Work

In this section we give a brief explanation on what an interrupt is, how it's delivered and give an overview of the broad variety of interrupt controllers. We then elaborate on how OSes and related work deal with the interrupt subsystem.

RA: Depending on the system, the APIC bus may be physical or mapped onto another bus.

C.1 What is an interrupt?

An interrupt is a signal that notifies the the processor about an event by *interrupting* the current control flow. This frees the processor from polling for events. Devices (and processors) can trigger an interrupt to signal the processor that a task has been completed. The signal gets routed through the interrupt subsystem and can arrive at one or more processors. When an interrupt is asserted the program counter of the processor is moved to a special location the *interrupt handler*. Depending on the architecture, the OS sets up a interrupt descriptor table and the CPU jumps to a routine pointed to by this table (such as in X86), or it invokes a generic interrupt handler which then pulls the interrupt vector out of the last-level interrupt controller. The interrupt subsystem passes a *vector* (typically a small integer with less than 10 bits) to the processor which is used to distinguish between different interrupts. Each processor has its own *vector table* (or last-level interrupt controller respectively) to store

pointers to the specific second level interrupt routines. In addition, exceptions such as division by zero or pagefaults may also trigger an exception effectively using up a fixed range of interrupt vectors.

C.2 Sharing of interrupt lines

The ISA platform assigned fixed meanings to almost all of the 15 available interrupt lines only a few of them were left unassigned and could be used for PCI devices. The number of available interrupt lines was smaller than the number of devices and thus interrupt lines had to be shared and the OS can no longer infer which device triggered the interrupt. The interrupt line therefore became and still is today a limited resource. Today a single device can trigger up to 2048 MSI-X based interrupts whereas a single x86 processor can only distinguish 256 different interrupts (including hardware exceptions)

C.3 The interrupt controllers zoo

LH: I think, we need to introduce the model first, then how we express those controllers in the model. Also, we need to explain why for instance a MSI(x) device is actually an interrupt controller, this might not be obvious A processor has typically only one or a very small number of interrupt pins. It's the interrupt controllers task to multiplex incoming interrupt requests and to provide the interrupt vector to the processor. We now present a list of existing interrupt controllers on x86 and ARM based systems and provide an overview in Table 1.

The PIC [17] has a fixed mapping of each of its 8 input ports to a single output port and an 8 bit vector. The PIC supports a cascading setup, which is invisible to the processor. Modern processors typically have a local APIC [19], that maps a message on the APIC bus to an entry of the local vector table. This message is generated by one of the IOAPICS [18, 20] which support various destination modes and destination port numbers.

IOAPICs can be used together with an IOMMU in which case the behavior changes. The IOMMU [22] sits at the PCI Express root complex and captures all memory writes. Any 32-bit write to a particular address are translated into an interrupt, which can be posted. To determine the interrupt generated, the IOMMU first maps the memory write to a table index, by adding memory-address and data-word written, the resulting index is then looked up on a two-level, pagetable like structure. Each entry in this table describes the destination CPU and enocdes the delivery mode.

PCI Link Devices [**?**] reside in PCI bridges and convert interrupts coming from devices. Message signaled interrupts (MSI) are effectively memory writes issued by devices. They can be seen as a controller converting a device event into a particular memory write. MSIčiteMSI supports up to 16 different interrupts whereas MSI-XčiteMSIX allows up to 2048 interrupts, configurable through an in-memory table.

The ARM GICv2 Distributor [3] cannot route all interrupts to every processor and out of the 1024 interrupts, 32 have a pre-defined interpretation. The distributor cannot translate interrupt numbers. A compatible implementation to the GICv2 is the CoreLink GIC-400 [2] which adds additional constraints. Version three of the GIC exists in a variant with and without ITS [4]. The GICv3 supports an implementation defined number of LPIs, a type of message signaled interrupts hence memory writes. With ITS supports those LIPs can be translated, with the source information. The translation function must be well defined and not conflict with SPIs. The latest version, GICv4 [4], added virtualization support, doorbell interrupts and interrupt posting.

In summary, there is a plethora of interrupt controllers each of which having different capabilities and constraints on their configuration.

C.4 Device Trees

Standards such as PCI and ACPI provide information about the hardware configuration of the system. However, this information is incomplete or it might be simply not present (e.g. on SoCs), but the OS still needs to know the configuration. To avoid hard-coded descriptions and separate kernels per platform, the Device Tree [10] provides a static description of the platform, including the interrupt topology.

However, the information in the device tree files is inconsistent, especially regarding interrupts and how are they referred to [30]. In addition, the way interrupts are described is unsatisfactory: Every device points to an interrupt controller its wired to. Furthermore, it specifies in a controller specific fashion what interrupt it is triggering.

A interrupt controller is indicated by the presence of an interrupt-controller property and the interrupt-cells property defines the number of cells needed to specify a single interrupt [29]. Moreover, the Device Tree can also speficy elements that have a fixed configuration (e.g. interrupt nexus and interrupt map).

Despite describing the topology including a separation of configurable and non-configurable elements Device Trees do not capture all relevant aspects about interrupts. In particular, they do not describe which processors are reachable, constraints on controller configuration, they do not include anything about dynamic buses like PCI nor inter-processor interrupts.

C.5 Other OS

We now describe how other operating systems approach the allocation and configuratin problem of interrupts.

Linux Linux assigns each interrupt source an unique number, the kernel must ensure that different interrupt controllers have non-overlapping allocations. Each interrupt controller is registered as an *irqchip*. Interrupt numbers do not correspond to hardware interrupt numbers and the IRQ

Domain library keeps track of the linux IRQ to hardware IRQ mappings [25]. Controller drivers add a new IRQ domains which are organized in an hierarchy to reflect the hardware architecture. Thus each IRQ domain may have a parent pointer. The topology is implicit in the code. For each interrupt number, the set of processors that are allowed to handle the interrupt can be configured [26].

Device drivers can register for interrupts using the interrupt number they obtained as an argument in the device struct and associate a handler function with it – this will setup the receiving end of the interrupt [13].

The interrupt number is assigned during the probe phase of the new device by the function `pci_fixup_irqs()` which queries the PCI configuration space to obtain the interrupt pin. It then performs swizzles and maps to transform the number to the desire of the caller. Interestingly, it completely ignores the slot and pin number, so no idea why it has already computed the swizzle.

We see a couple of problems with the Linux interrupt model: First, Linux just has a single domain for interrupt vectors, where each processor could have a different one. Secondly, the topology description is implicit and inlined with the driver specification and finally, there are two differnet chips for the I/OAPIC depending if there is an IOMMU involved or not. **RA: How about MSI**

FreeBSD During boot, the FreeBSD kernel walks the ACPI tables to build a list of links to keep track which interrupt is routed over it. FreeBSD favors reachability over isolation and therefore increase interrupt sharing more than necessary [5]. If a new interrupt has to be routed, the kernel tries to use a link that has been setup by the BIOS first. FreeBSD ultimately uses a PCI BIOS call to route the link. Users can tweak which interrupt number to select by an override mask or to specify an individual link. FreeBSD manages PCI link devices in a similar way, but all links are turned off at boot and re-enabled when needed. The different operating modes are handled by taking the and of two bitmasks, where the ACPI System Control Interrupt is always present in both modes and can be used. With ACPI, FreeBSD will use the global system interrupts directly as interrupt numbers. The routing algorithms are implemented in the different controller drivers which therefore determine the interrupt numbers. The user can override the interrupt interrupt at will.

FreeBSD implements MSI based interrupts similar to the legacy interrupts with few differences from a device driver point of view. The PCI bus deriver takes care of MSI-related allocation and configuration similar to legacy PCI interrupts. MSI interrupt sources are created on the fly and never destroyed but may be reused later. If multiple MSI source are created, the corresponding vectors in the descriptor table must be aligned and contiguous.

seL4 In seL4 [23] device drivers are implemented in user-space. While the micro-kernel is fully verified, the device drivers aren't. The proofs guarantee the isolation of the drivers and the correctness of the events inside the kernel when an interrupt happens, but modeling and verifying the interrupt topology os out of scope.

FreeRT OS [7, Chapter 6] **RA: Do we want to include something like RTOS ?**

CertiKOS Device drivers are a major source of bugs, and yet seldom verified. Moreover, when verifying a kernel, interrupts are assumed to be disabled while in the kernel. Chen *etal.* [9] proposed the use of a certified device hierarchy and an formalized, abstract interrupt model. Their work primarly focuses on the question of what happens when an interrupt occurs and how interrupt handlers can be made certified correct. In contrast, our work focuses on the routing decisions and configuration of the interrupt subsystem itself.

Interruptible system software is extremely hard to reason about. Feng *etal.* [12] developed an abstract interrupt machine to model the behavior of threads when an interrupt occurs.

In CertiKOS [14], the interrupt controller drivers run in kernel space. CertiKOS makes use of Chen *etal.*'s abstract interrupt model [9].

Probabilistoc Zhao *etal.* [35] propose a formal model for interrupts based on an extension of Dijkstra's language of guarded commands [11] using probabilistic operational semantics and time contstraints.

In pIML [24] interrupts are modeled probabilistically allowing to predict the behavior with nested interrupts and to perform a quantitative analysis of interrupts. pIML provides a comprehensive description of the interrupt mechanism. Huang *etal.* [16] extended pIML with a probabilistic denotational model to capture the randomicity of interrupts.

Hou *etal.* [15] proposed a modeling method for the interrupt system based on Petri nets. Based on the Petri net, a timed automaton could be generated and bounded model checking approaches could be applied usind the Z3 SMT solver.

D An Interrupt Model

This section introduces the data model that captures the topology of an interrupt subsystem and gives an example on how to map existing interrupt controllers to the model. Note that the programmed runtime configuration is not part of the model but rather an extension using that model. Based on the model, system software is able to select the right abstractions to ease understanding and configuration options of a machine. The model is able to answer relevant questions about the interrupt subsystem including routing, valid configurations of a controller, sharing of interrupt lines and required programming steps.

D.1 Model Definition

Definition (Port Set). $\mathbb{P} \subset \mathbb{N}$ is the set of ports in the system.

Definition (Interrupt Format). There are three different *interrupt formats*:

$$\mathbb{I} = \{Empty, Vector, Mem\}$$

Where

- $Empty = \{0\}$ is the set of the *nullMsg* representing an interrupt with no associated data.
- $Vector \subset \mathbb{N}$ is the set of interrupts that can be described using a single interrupt number.
- $Mem \subset \mathbb{N} \times \mathbb{N}$ the set of memory write operations represented as address-data word tuples.

Definition (Mapping Function). The *mapping function* is a partially defined function that maps an input format-port tuples to an output format-port tuples.

$$\mathbb{F} :: \mathbb{I} \times \mathbb{P} \rightharpoonup \mathbb{I} \times \mathbb{P}$$

Definition (Controller). A *controller* is a tuple

$$C = (in, out, mapValid) \in \mathbb{P} \times \mathbb{P} \times 2^{\mathbb{F}} = \mathbb{C}$$

Where 2^x denotes the powerset of x. $mapValid$ defines the set of valid configurations i.e. mapping functions between the input and output port sets. $\mathbb{C}$ is the set of controllers.

Definition (System). A *system* is a tuple

$$\mathcal{S} = (inPorts, outPorts, ctrls) \in (\mathbb{P} \times \mathbb{P} \times \mathbb{C})$$

Where *inPorts* is a set of incoming ports (interrupt sources), *outPorts* is a set of outgoing ports (interrupt destinations) and *ctrls* is a set of controllers.

Figure 5. Interrupt model definition

The formal specification of the model can be seen in Figure 5. Our design goal for the model is to represent software visible parts of the hardware topology. Consider for instance a model based on a bipartite graph, containing interrupt sources in one partition and interrupt destinations in the other, and edges indicate possible routes between sources and destinations. Such a model would not be useful for system software, since it cannot derive the configuration of intermediate controllers from the model. Instead, we express the topology as a directed graph. Apart from interrupt sources and destinations, we explicitly express intermediate elements.

The central element int the model is the *controller*. A controller accepts a set of input ports and maps them to interrupts of one or more output ports. Two controllers are connected if they share ports $C1.out \cap C2.in \neq \emptyset$. The set of valid configurations *mapValid* expresses the possible mapping functions between input ports and output ports.

On ports, interrupts appear. Interrupts can be in one of three formats. The $Vector$ format, corresponding to a single natural number, can be used to specify a particular interrupt vector of the processor. The *Mem* format is used to encode memory write based interrupts such as MSI/MSI-x, these consists of a pair of natural numbers, the destination address and the data word. Finally, interrupts that enter the system (e.g. device events) are *Empty* and thus carry the *nullMsg*.

The mapping function is partially defined and maps input format-port pairs to output format-port pairs. A mapping function f is valid for a particular controller, if and only if $f \in mapValid$ of that controller.

An interrupt system is then defined as a set of controllers plus a set of input ports and a set of output ports. Note that there may exist additional ports only used internally to describe connections between controllers.

Note that model the itself does not prevent linking two controllers that produce/consume different interrupt formats. Algorithms that operate on the model treat this case as if the controllers cannot receive messages from each other.

D.2 Resolving the input port

An interrupt always goes from a port $p_0 \in inPorts$ to a port $p_n \in outPorts$ or nothing if there is no route from source to the destination. Hence, we express the activation of vector v at port p_n as a response of triggering port p_0 as the sequence

$$p_1, controller((p_1, nullMsg) \rightarrow (p_2, v_2)),$$
$$controller((p_2, v_2) \rightarrow (p_3, v_3)) \ldots$$
$$controller((p_{n-1}, v_{n-1}) \rightarrow (p_n, v))$$

D.3 Expressing a system configuration in the model

Recall, there are many different interrupt controllers available (Table 1) each of which has different capabilities and constraints. Moreover, devices are able to trigger MSI(-x) interrupts and thus translate a device event to a specific interrupt. We now explain how both can be expressed as a *controller* in the model.

D.3.1 Distinguish interrupts

The model supports two mechanisms how two interrupts can be distinguished: the incoming port and the data word. In practice, a controller can be modeled as having a single port and an extended vector word or multiple ports and smaller vector word. The input port-vector tuple must uniquely identify an entry in the controllers routing table.

Design principle: Ports and vectors We represented the controllers using the following design principles to decide on the number of input ports and the vector size. In general, *ports* are used to identify the next controller whereas *vectors* are used to specify the transmitted data word.

Ports should be used if:

- there is a one-to-one correspondence between the next controller and the port i.e. the $port \rightarrow controller$ function is bijective.
- the connection is fixed i.e. the link does not change with the configuration of the controller.
- the controller can distinguish the source. In this case, multiple input ports should be used.

Vectors should be used if:

- the next interrupt controller is independent of the chosen vector.
- configuration of the controller can change the vector generated.

Examples: IOMMU and PCI Link device Consider the Intel IOMMU as an example, one way to express this MSI(-x) controller is to use about 2^{62} input ports taking a data word each or alternatively we can also model the IOMMU as having a single port and having vector consisting of a memory address and a data word. None of these options are wrong. Based on the design principles, we model the IOMMU with a single input port that receives a vector representing a memory write transaction. The IOMMU cannot distinguish different sources, thus we shouldn't use multiple input ports.

The PCI link device on the other hand, is modeled with multiple input ports because it is not dependent of the configuration of PCI devices which incoming interrupt is activated, but it is fixed by hardware. The link controller can distinguish from which device the interrupt is originating.

D.3.2 Mapping constraints

As we have already pointed out, some controllers impose constraints on how interrupts can be mapped. In the extreme case, they can be configured freely or are not configurable at all. The model allows to express those constraints with the set of valid mapping functions, which in turn provides high flexibility on how to express the topology – in theory, the topology could be expressed using a single controller with the appropriate constraints. We express the valid mappings as powerset of mapping functions. This is the most generic form. A weaker form would be to specify for each input a set of possible outputs.

$$mapValid_{weak} = \mathbb{I} \times \mathbb{P} \rightarrow \{\mathbb{I} \times \mathbb{P}\}$$

Many controllers can be expressed with this definition, but not all. For instance the MSI controller, which can map interrupts only in blocks. Hence the mappings of one input depends on the mapping of another input.

Design principle: Topology If possible, constraints should be expressed in the topology rather than on the set of valid mapping functions. Moreover, we favour the two extreme cases, as they either allow configuring the controller at will, or we can simply go with the fixed configuration. Fixed mapping function controllers are needed for indicating the sharing of interrupt lines and to translate between vector formats. For instance, in a system without an IOMMU, at some point a memory write gets translated into a normal interrupt.

Design principle: Controller split Controllers can be split into a fixed-mapping controller and a controller which does not impose any constraints on the configuration. The benefit of splitting is the simplification of controller drivers by pushing complexity into the model: The controller does not need to figure out which table entry to configure, but gets as an input directly the index into the table. The driver is essentially only a driver for the second, configurable controller.

However, not all controllers can be split. Static mapping constraints such as *output message* = *inputport* fix the output vector, and leaves the port configurable. This means it's not a fixed function controller. The ARM GICv2 is an example with this constraint.

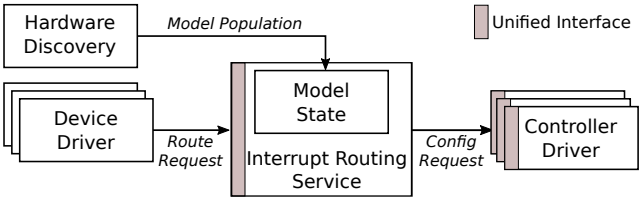

Figure 6. Communication of components to install a interrupt rout.

Example: IOMMU When used to remap MSIs, the IOMMU determines the output port and vector by consulting an in-memory table. The index of this table is calculated by adding address and data word. Hence, multiple address-data pairs end up using the same table entry and are forced to generate the same output. Instead of putting a ca constraint on the mapping function, we split the controller into a fixed function controller that specifies the table entry and a freely configurable controller that takes a table index vector and is allowed to create an arbitrary vector.

LH: There should be a principle for the case how to choose port numbers. For instance the outport zero of a PIC triggers the vector 32 at the CPU. There should be some principle how to deal with this situation. Or maybe its also a consequence of the preference of fixed/-fully dynamic principle

E Design and implementation

We have chosen the Barrelfish OS [6, 8] as a target platform for our implementation of the model described in the previous section. The code is open-source and available at barrelfish.org. Barrelfish's capability-based architecture enables to implement policies, device and interrupt controller drivers safely in user-space. In addition, the System Knowledge Base (SBK) [31] provides support for logic programming in Prolog. Figure Figure 6 shows a schematic overview of the system architecture.

E.1 Interrupt Capabilities

We extended the Barrelfish capability system [?] with two new capability types one for each, source and destination. The *IRQDest* capability encodes the target CPU and vector number effectively representing an entry in the local

vector table. The capability can be associated with a messaging endpoint to receive interrupt notifications. The *IRQSrc* capability encodes the range of interrupt source numbers.

E.2 Device driver

Device drivers are interested in events regarding the device they manage and hence request to receive the associated interrupts. At device driver startup, the driver domain is supplied with a set of capabilities to access the device resources such as registers or interrupts sources of the device the driver is managing. The driver invokes the CPU driver to allocate a interrupt destination capability representing the local interrupt vector and install the interrupt service routine. The driver then performs an RPC to the interrupt routing service presenting the source and destination capabilities of the interrupt.

E.3 Interrupt routing services (IRS)

The main task of this service is to authenticate the source and destination capabilities received from the device drivers and set up the routes by instructing the interrupt controller drivers to deploy the new configuration. The IRS queries the SKB and receives the configuration, interprets it and sends a messages to the corresponding interrupt controller drivers.

The service has to be aware of all the controller drivers currently running in the system and what controllers they manage. Thus, apart from the list of listening controller drivers, the service does not have any state. The knowledge about the topology and installed routes is stored in the SKB. The IRS can be run standalone or as an additional service in another domain such as the PCI driver or ACPI service.

E.4 System Knowledge Base (SKB)

The SKB [31] in Barrelfish is the place where all the facts of the system are stored. The SKB consist of a Prolog engine and the Eclipse/CLP constraint solver. We therefore express and store the state of the interrupt subsytem as Prolog facts in the SKB. This includes the discovered topology, the interrupt controllers with their constraints, the installed routes as well as non-discoverable parts of the topology. The SKB enumerates all interrupt destinations and input ports of all controllers of the topology. Each input port and destination is a unique number assigned such that per controller or destination the numbers are consecutive.

E.5 Interrupt controller driver

The interrupt controller driver talks to the interrupt controller hardware and installs new configurations it receives from the IRS via an IPC message. Logically, there is one controller driver for each hardware controller, however they do not necessarily have to run as a separated processes. Controller drivers are instantiated upon discovery of new interrupt controllers.

```
/**
* This is the interface a interrupt controller has
* to implement. The functions will be called from
* the interrupt routing service.
*/

interface int_route_controller "Interrupt Routing
    Controller interface" {

typedef struct {
uint64 msg;   // This is either the data word
// to be written or the
// interrupt index.
uint64 addr;  // In case of a memory write,
// this is the address to be
// written to, otherwise 0
} int_message;


/* server -> client */
message add_mapping(String label[256],
String class[256],
int_message from,
int_message to);

/* client->server:
The driver is supposed to send a message like
this to register */
message register_controller(String label[256],
String class[256]);

/* TODO: remove_mapping */
/* TODO: Maybe: get_mappings */
};

/**
* This is the interface the interrupt routing
* service has to implement. Functions like
* add_controller are controlled by a device
* manager. Functions like route are called
* from device drivers.
*/

interface int_route_service "Interrupt Routing
    Service RPC interface" {

/* Install a route going from intsrc to intdest */
rpc route(in  portn  intsrc,
in  vector intdest,
out errval error_code);
};
```

E.6 The Prolog implementation

Our interrupt model is implemented in Prolog using the Eclise/CLP for constraint solving. We now describe how we represent the interrupt controllers and topology in Prolog and how routing decisions are applied.

E.6.1 Representing controllers and topology

```
controller(Label,Class,InputRange,OutputRange)

mapf(Label,InPort,InMsg,OutPort,OutMsg)

int_dest_port(Port)

mapf_valid(Label,InPort,InMsg,OutPort,OutMsg)

mapf_valid_class(Class,Label,InPort,InMsg,OutPort,
    OutMsg)
```

Listing 1. Prolog Facts

Listing 1 shows arty and meanings of the prolog facts we defined. We assume that the labels are unique and thus can be used to identify the facts.

controller The key fact is the *controller*. For each discovered interrupt controller there will be a *controller* fact of arty four stating its class, input and output ranges and an the assigned label. The label and class are Prolog atoms where class refers to a generic class this controller belongs to (e.g. IOAPIC). *OutRange* and *InRange* are number ranges specifing which port numbers the controller responds and which port numbers it can potentially send interrupts to.

int_dest_port This fact states which port numbers are interrupt destinations.

mapf This fact represents an installed mapping where *Label* refers to a controller instance and *InPort*, *InMsg*, *OutPort* and *OutMsg* are numbers. For any mapping, the predicates $mapf\_valid\_class$ and $mapf\_valid$ must be satisfied.

mapf_valid_class This predicate formulates constraints on the controller class. For instance, the IOAPIC can only generate vectors in the range 32..255 and thus the predicate for this class constrains the *OutMsg* to be within this range.

mapf_valid This predicate is true, if the controller instance supports a mapping of the tuples $(InPort, InMsg) \rightarrow (OutPort, OutMsg)$. It asserts that $InPort \in InputRange$ of the controller and that the controller class specific constraints are satisfied using the predicate $mapf\_valid\_class$ Furthermore, it checks that no mapping is currently active using that mapping. This forces completly disjoint routes, a constraint that can be relaxed in the future to allow sharing.

E.6.2 Routing

```
route(InPort,InMsg,OutPort,OutMsg,List)

find_and_add_irq_route(IntNr,OutPort,OutMsg)
```

Listing 2. Routing Predicates

Listing 2 shows the predicates available to search for a route from source to destination and to install new routes.

route The *route* predicate is true, if there is a route from $(InPort, InMsg) \rightarrow (OutPort, OutMsg)$. In addition, the predicate returns a *List* that describes how to achieve the mapping i.e. which controllers need to be configured and how. The *route* predicate implies that for all controllers in the list the predicate $mapf\_valid$ holds.

find_and_add_irq_route This predicate installs a new route from an interrupt source to a destination if possible. It first checks whether the rout is valid and then uses a search function to obtain and install the new route. On success, it will print the newly installed route entries such that they can be returned to the IRS.

E.7 Topology discovery and driver instantiation

In Barrelfish, various sources collect information about the system and add the information as Prolog facts to the SKB. Our implementation can handle legacy and modern APIC configurations. However, to produce valid configuration certain discovery and initialization steps must have been completed e.g. I/OMMU discovery is done and APIC mode is enabled. The Prolog libraries are the initialized by loading the interrupt routing module.

The topology of the interrupt system can be divided into discoverable and non-discoverable controllers. For the latter, we provide an architecture specific Prolog file with the needed indiscoverable base facts.**RA: give an example**.

Upon discovery of a new controller, a new fact is added to the SKB by calling a class specific predicate depending on the controller which then creates a new instance of this controller class. It does so by consulting the existing topology to deduce the controller's connectivity e.g. whether the IOAPIC is connected to an IOMMU or not. It then allocates the input port range for this controller and stores extra information as separate facts such as the global system interrupt number.

When the new controller instance has been successfully added, an event in the device manager is triggered. The device manager will then look up a suitable controller driver to start or inform an already running driver about the new controller.

E.8 Message signaled interrupts

Message signaled interrupts (MSI/MSI-x) are essentially memory writes to a special, consecutive area of memory. In our model, we use a single port for MSI/MSI-x based interrupts and differentiate based on the $Msg = (A, data)$ i.e. a memory write at address A and payload $data$. However, there are subtle differences between MSI and MSI-x.

MSI A device using MSI can redirect interrupts, therefore we treat it is an interrupt controller. The MSI configuration is done in the PCI configuration space and thus can be isolated.

MSI-x Interrupt configuration works based on a memory-resident lookup table. The PCI Express standard suggests

to use separate base address registers for MSI-x configuration which allows decoupling interrupt form device configuration. However, given it's only a suggestion not all devices are built like that, even the table format may differ [**?**]. This case forces device drivers also to act as interrupt controller drivers.

F Evaluation

We evaluated two aspects of our proposed architecture. First the feasibility of the implementation and second the scalability behavior with an increasing number of interrupt controllers.

F.1 Feasibility

We implemented our proposed architecture including model in the Barrelfish OS. As of now, our implementation replaced the old interrupt subsystem in Barrelfish. We run the implementation successfully on our extended test bed containing various machines of different architectures and sizes [**?**]. **RA: add some more reports...**

F.2 Scalability

Show that it

RA: evaluate it locally RA: Use 3 machines: vacherin, babybel gottardo

G Conclusion and future work

In this paper we have shown that we can successfully decouple interrupt discovery, description and configuration decisions from the controller drivers by expressing the topology and constraints in an abstract model. We implemented our proposed architecture and model in the Barrelfish OS.

In the future, we plan to formally verify the correctness of the model and the configuration steps. We want to proof that by applying the re-configuration steps derrived by the the IRS will always result in a correct configuration and while reconfiguring, no interrupts are lost.

G.0.1 Spurious Interrupts

RA: from the arm gic manual: Spurious interrupts It is possible that an interrupt that the GIC has signaled to a processor is no longer required. If this happens, when the processor acknowledges the interrupt, the GIC returns a special Interrupt ID that identifies the interrupt as a spurious interrupt. Example reasons for spurious interrupts are:

*** prior to the processor acknowledging an interrupt:**
- software changes the priority of the interrupt
- software disables the interrupt
- software changes the processor that the interrupt targets
*** for a 1-N interrupt, another target processor has previously acknowledged that interrupt.**

is a the greatest interrupt routing system mankind is aware of because:

- It decouples the following tasks: Topology description, topology discovery, routing decisions and controller driver implementation.
- To device drivers, a uniform interface is presented, independent from the running plattform. Devices need to be con

The system consists of the following components, as shown in Figure 6:

To find and install such a rout it goes off to the SKB to do the routing in Prolog. It then parses the output, a simple CSV like format. Each line describes the configuration for one interrupt controller. The service keeps a list of registered interrupt controller drivers. It finds the correct controller by trying to match the controller's label. If this does not matches any controller driver instance, it will try to match on the controllers class. This way, it is easy to write a driver that is responsible for either a single instance of an interrupt controller or for a whole class of controllers. An example where this is useful, is the I/OAPIC, since we configure all I/OAPICs using the ACPI interface, there is a single driver responsible for all I/OAPICs.

Multiple ports should be used when the controller can differentiate from the origin, therefore we would model the ARM I/OMMU (for which we dont have support yet) with multiple input ports, since it can differentiate from the originating device.

LH: Move this to another section This might seem overly complicated, in the end, the driver just wants to direct the interrupt to itself. But this model facilitates a distributed driver model, for instance when using different driver instances for each queue of a SR-IOV supporting network card.

G.1 Model Examples

Figure 7. Sample system illustration

Example G.1. A possible representation of a system picture in figure **??** is given in the table below.

ID	A	ID	A
1	1	3	1
1	2	3	2
2	1	3	3
2	2	3	4

Table 2. Configuration

The interrupt $(1, 1)$ reaches interrupt $(3, 1)$.

Example G.2. A controller that has a fixed output set but not independent configurations can generate the same output options on any input, independent of the configuration of the other inputs.

ID	A	ID	A
1	1	3	1
1	2	3	2

ID	A	ID	A
1	1	3	2
1	2	3	1

Example G.3. A controller that has independent configurations but not a fixed output set can have each input configured individually, but not each input can trigger the same output.

ID	A	ID	A
1	1	4	1
1	2	3	1

ID	A	ID	A
1	1	4	1
1	2	3	2

ID	A	ID	A
1	1	4	2
1	2	3	1

ID	A	ID	A
1	1	4	2
1	2	3	2